```
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
  *        U. S.   P A T E N T   T E X T   F I L E                *
  *                                                               *
  *  THE WEEKLY PATENT TEXT AND IMAGE DATA IS CURRENT             *
  *  THROUGH AUGUST 31,1999                                  *
  *                                                               *
  *                                                               *
  * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

=> s 712/?/ccls

L1        5147 712/?/CCLS

=> s (execution (w) stack?)

         63102 EXECUTION
        172617 STACK?
L2          47 (EXECUTION (W) STACK?)

=> s l1 and (exception? or interrupt?)

        154378 EXCEPTION?
        228578 INTERRUPT?
L3        2779 L1 AND (EXCEPTION? OR INTERRUPT?)

=> s ((execution (w) stack?) (p) (exception? or interrupt?))

         63102 EXECUTION
        172617 STACK?
        154378 EXCEPTION?
        228578 INTERRUPT?
L4          11 ((EXECUTION (W) STACK?) (P) (EXCEPTION? OR INTERRUPT?))

=> d kwic l4 1-11

US PAT NO:      5,933,635 [IMAGE AVAILABLE]             L4: 1 of 11

SUMMARY:

BSUM(2)

 This . . . and Apparatus for Performing Byte-Code Optimization During
Pauses, U.S. patent application Ser. No. 08/944,335 (Atty. Docket No.
SUN1P150/P2300), entitled "Mixed Execution Stack and
Exception Handling," U.S. patent application Ser. No. 08/944,326
(Atty: Docket No. SUN1P152/P2302), entitled "Method and Apparatus for
Implementing Multiple Return Sites,". . .

US PAT NO:      5,590,332 [IMAGE AVAILABLE]             L4: 2 of 11

DRAWING DESC:

DRWD(10)
 interrupting any of said CPS-converted subprograms when the
  execution stack pointer is beyond a limit point of the stack
  buffer;

DRAWING DESC:

DRWD(18)
 interrupting a CPS-converted subprogram when the execution
  stack pointer is beyond a limit point of the stack buffer;
```

DRAWING DESC:

DRWD(27)
interrupting a CPS-converted subprogram when the **execution stack** pointer is beyond a limit point of the stack buffer;

DRAWING DESC:

DRWD(35)
interrupting a CPS-converted subprogram when the **execution stack** pointer is beyond a limit point of the stack buffer;

CLAIMS:

CLMS(1)

I . . .
subprograms into continuation-passing style (CPS) in the stack-oriented language;
determining the extent and limit points of a stack buffer on the **execution stack** of the stack-oriented language, which buffer is capable of holding a plurality of invocation stack frames;
commencing the execution of the application such that the initial invocation stack frame is within the limits of the stack buffer;
interrupting any of said CPS-converted subprograms when the **execution stack** pointer is beyond a limit point of the stack buffer;
initiating an evacuation process to transport out of the stack buffer into the heap any live data objects found in the stack buffer;
contracting the **execution stack** of the stack-oriented language back to the beginning of the stack buffer; and
resuming the execution of the **interrupted** CPS-converted subprogram.

CLAIMS:

CLMS(6)

6. . . .
subprograms into continuation-passing style (CPS) in the stack-oriented language;
determining the extent and limit points of a stack buffer on the **execution stack** of the stack-oriented language, which buffer is capable of holding a plurality of invocation stack frames;
commencing the execution of the mutually recursive subprograms such that the initial invocation stack frame is within the limits of the stack buffer;
interrupting a CPS-converted subprogram when the **execution stack** pointer is beyond a limit point of the stack buffer;
initiating an evacuation process to transport out of the stack buffer into the heap any live data objects found in the stack buffer;
contracting the **execution stack** of the stack-oriented language back to the beginning of the stack buffer; and
resuming the execution of the **interrupted** CPS-converted subprogram.

CLAIMS:

CLMS(11)

11. . . .
so that continuation closure objects are local stack-allocated objects;
determining the extent and limit points of a stack buffer on the **execution stack** of the stack-oriented language, which buffer is capable of holding a plurality of invocation stack frames;
commencing the execution of the. . . . that the initial invocation stack frame is within the limits of the stack buffer;

capturing one of the stack-allocated continuation closure objects;
interrupting a CPS-converted subprogram when the **execution stack** pointer is beyond a limit point of the stack buffer;
initiating an evacuation process to transport out of the stack buffer into the heap any live data objects including live continuation closure objects found in the stack buffer;
contracting the **execution stack** of the stack-oriented language back to the beginning of the stack buffer; and
resuming the execution of the **interrupted** CPS-converted subprogram.

CLAIMS:

CLMS(15)

15. . . .
so that continuation closure objects are local stack-allocated objects;
determining the extent and limit points of a stack buffer on the **execution stack** of the stack-oriented language, which buffer is capable of holding a plurality of invocation stack frames;
commencing the execution of the mutually recursive subprograms such that the initial invocation stack frame is within the limits of the stack buffer;
interrupting a CPS-converted subprogram when the **execution stack** pointer is beyond a limit point of the stack buffer;
initiating an evacuation process to transport out of the stack buffer. . . generation of a garbage-collected heap any live data objects including live continuation closure objects found in the stack buffer;
contracting the **execution stack** of the stack-oriented language back to the beginning of the stack buffer; and
resuming the execution of the **interrupted** CPS-converted subprogram.

US PAT NO: 5,530,870 [IMAGE AVAILABLE] L4: 3 of 11

SUMMARY:

BSUM(17)

Because . . . that is conventionally consumed in both performing the individual returns between each calling and called subprogram, and in manipulating the **execution stack** in order to effect the individual returns, is saved. System throughput and performance are markedly improved in modular implementations as a consequence. Yet the method and arrangement use the conventional **execution stack** in order to effect the series of invocations and the return therefrom, as opposed to using an **exceptional**, special, off-stack mechanism, as is done in some prior art arrangements. Consequently, unlike in the prior art, a subprogram can. . .

US PAT NO: 5,522,072 [IMAGE AVAILABLE] L4: 4 of 11

SUMMARY:

BSUM(17)

Because . . . that is conventionally consumed in both performing the individual returns between each calling and called subprogram, and in manipulating the **execution stack** in order to effect the individual returns, is saved. System throughput and performance are markedly improved in modular implementations as a consequence. Yet the method and arrangement use the conventional **execution stack** in order to effect the series of invocations and the return therefrom, as opposed to using an **exceptional**, special, off-stack mechanism, as is done in some prior art arrangements. Consequently, unlike in the prior art, a subprogram can. . .

US PAT NO: 5,412,717 [IMAGE AVAILABLE] L4: 5 of 11

DETDESC:

DETD(88)

Turning . . . block 336 the execution of the current program either starts or resumes execution upon being placed on top of the **execution stack**. The processing which occurs in blocks 336 through 410 includes operations which are conventionally preformed to execute a program. Processing. . . . loading) the state of various registers to reflect their state at the point in time when the program was last **interrupted** (or initialized). Additionally, system status information is restored, e.g., such as stack pointers, etc., depending upon the particular system environment. . .

US PAT NO: 5,311,591 [IMAGE AVAILABLE] L4: 6 of 11

DETDESC:

DETD(88)

Turning . . . block 336 the execution of the current program either starts or resumes execution upon being placed on top of the **execution stack**. The processing which occurs in blocks 336 through 410 includes operations which are conventionally preformed to execute a program. Processing. . . . loading) the state of various registers to reflect their state at the point in time when the program was last **interrupted** (or initialized). Additionally, system status information is restored, e.g., such as stack pointers, etc., depending upon the particular system environment. . .

US PAT NO: 5,109,329 [IMAGE AVAILABLE] L4: 7 of 11

ABSTRACT:
A . . . slot of a uniprocessor, and by minimally modifying the uniprocessor's operating system. At initialization, one routine (FIG. 5) redirects slave **interrupt** vectors (200) to point to a common **interrupt** handler (FIG. 12). Before a process executes on the slave processor, another routine (FIGS. 9 and 10) corrupts **execution stack** bounds (217, 218) of the process. A non-**interrupt** operating system call during execution of the process causes an automatic firmware check (FIG. 3) of the **execution stack** pointer (203) against the stack bounds. Occurrence of an **interrupt** or encounter of a stack **exception** results in suspension of process execution and invocation of the **interrupt** handler or a slave stack **exception** handler (FIG. 11), respectively. Each handler calls a slave delete routine (FIG. 15) to restore the process' stack bounds to. . . processor, process execution resumes at the point of suspension, and the operating system service required by the system call or **interrupt** is carried out.

DETDESC:

DETD(10)

The system call mechanism uses the **execution stack** of the present process; that is, a normal **exception** handler or a function called via a GATE instruction uses for its execution the **execution stack** of the process that was executing when the **exception** or the GATE call occurred.

DETDESC:

DETD(12)

In . . . entry of a privileged function, i.e., before executing a transfer to the privileged mode upon the occurrence of a normal **exception** or a GATE request, the system call mechanism checks the present **execution stack** pointer value against the **execution stack** boundary values that are stored in the process control block of the presently-executing process, in the manner shown in FIG.. . . within the specified bounds, at step 306; if the stack pointer does not fall within the specified bounds, a stack **exception** is generated, at step 308. The microprocessor performs the check automatically, either directly in hardware or by execution of a. . .

DETDESC:

DETD(13)

If . . . stack pointer is found to fall within the specified bounds upon the occurrence of a GATE call or a normal **exception**, the processor handles the normal **exception** or GATE request within the process in which it occurred: the processor status word and the program counter of the process that was executing when the system call mechanism was activated are stored on that process' **execution stack**, the stack pointer is incremented, and the program counter and processor status word of the called function are loaded into. . . activities are likewise performed automatically, either by hardware or by execution of a micro-instruction sequence. Illustratively, GATE calls and normal **exceptions** have their own separate micro-instruction sequences.

DETDESC:

DETD(14)

The process switch mechanism is used by **interrupts** and "non-normal" **exceptions** including the stack **exception**. The process switch mechanism uses a different **execution stack** for the old and the new processes. Thus, for example, the stack **exception** handler process has its own **execution stack** different from the **execution stack** of the excepted-to process. Similarly, the **interrupt** handler process has its own **execution stack** different from the **execution stack** of the **interrupted** process. Because a different **execution stack** is used for each **interrupt** handler and non-normal **exception** handler, the **execution stack** bounds check is not performed upon the occurrence of an **interrupt** or a non-normal **exception**.

DETDESC:

DETD(21)

The automatic invocation is basically accomplished as shown in FIG. 4 . **Execution stack** bounds 217, 218 stored in process control block 200 of a process are given an improper value, at step 450,. . . during an attempt to enter privileged execution mode, at step 452, via a GATE call or occurrence of a normal **exception**. The failure of the check results in invocation of the stack **exception** handler process, at step 455. Also, at system initialization, **interrupt** and **exception** process control blocks are set up for slave processor 25 in its private memory 101, and values therein for handlers of **interrupts** and non-normal **exceptions** that may occur on slave processor 25 are redirected, at step 456, to the value of an error-handler process that is a duplicate of the stack **exception** handler process for purposes of this application. (An alternative to using private on-board memory is to duplicate virtual-to-physical translation tables,. . . as to provide each processor with different, exclusive, virtual-to-physical translations for certain ranges of virtual addresses.) Upon occurrence of an **interrupt** or non-normal **exception** on slave processor 25, at

step 457, these values cause invocation of the handler process, at step 458. The stack **exception** and error handler processes of the slave processor 25 are communication processes that restore, at step 459, to a proper. . . been redirected at step 456, a conventional handler would have been invoked at step 458 that would have processed the **interrupt** or condition, at step 463, as is done on master processor 12.)

US PAT NO: 5,003,466 [IMAGE AVAILABLE] L4: 8 of 11

ABSTRACT:
A . . . slot of a uniprocessor, and by minimally modifying the uniprocessor's operating system. At initialization, one routine (FIG. 5) redirects slave **interrupt** vectors (200) to point to a common **interrupt** handler (FIG. 12). Before a process executes on the slave processor, another routine (FIGS. 9 and 10) corrupts **execution stack** bounds (217, 218) of the process. A non-**interrupt** operating system call during execution of the process causes an automatic firmware check (FIG. 3) of the **execution stack** pointer (203) against the stack bounds. Occurrence of an **interrupt** or encounter of a stack **exception** results in suspension of process execution and invocation of the **interrupt** handler or a slave stack **exception** handler (FIG. 11), respectively. Each handler calls a slave delete routine (FIG. 15) to restore the process' stack bounds to. . . processor, process execution resumes at the point of suspension, and the operating system service required by the system call or **interrupt** is carried out.

DETDESC:

DETD(10)

The system call mechanism uses the **execution stack** of the present process; that is, a normal **exception** handler or a function called via a GATE instruction that uses for its execution the **execution stack** of the process that was executing when the **exception** or the GATE call occurred.

DETDESC:

DETD(12)

In . . . entry of a privileged function, i.e., before executing a transfer to the privileged mode upon the occurrence of a normal **exception** or a GATE request, the system call mechanism checks the present **execution stack** pointer value against the **execution stack** boundary values that are stored in the process control block of the presently-executing process, in the manner shown in FIG.. . . within the specified bounds, at step 306; if the stack pointer does not fall within the specified bounds, a stack **exception** is generated, at step 308. The microprocessor performs the check automatically, either directly in hardware or by execution of a. . .

DETDESC:

DETD(13)

If . . . stack pointer is found to fall within the specified bounds upon the occurrence of a GATE call or a normal **exception**, the processor handles the normal **exception** or GATE request within the process in which it occurred: the processor status word and the program counter of the process that was executing when the system call mechanism was activated are stored on that process' **execution stack**, the stack pointer is incremented, and the program counter and processor status word of the called function are loaded into. . . activities are likewise performed automatically, either by hardware or by execution of a micro-instruction sequence. Illustratively, GATE calls and normal

exceptions have their own separate micro-instruction sequences.

DETDESC:

DETD(14)

The process switch mechanism is used by **interrupts** and "non-normal" **exceptions** including the stack **exception**. The process switch mechanism uses a different **execution stack** for the old and the new processes. Thus, for example, the stack **exception** handler process has its own **execution stack** different from the **execution stack** of the excepted-to process. Similarly, the **interrupt** handler process has its own **execution stack** different from the **execution stack** of the **interrupted** process. Because a different **execution stack** is used for each **interrupt** handler and non-normal **exception** handler, the **execution stack** bounds check is not performed upon the occurrence of an **interrupt** or a non-normal **exception**.

DETDESC:

DETD(21)

The automatic invocation is basically accomplished as shown in FIG. 4 . **Execution stack** bounds 217, 218 stored in process control block 200 of a process are given an improper value, at step 450,. . . during an attempt to enter privileged execution mode, at step 452, via a GATE call or occurrence of a normal **exception**. The failure of the check results in invocation of the stack **exception** handler process, at step 455. Also, at system initialization, **interrupt** and **exception** process control blocks are set up for slave processor 25 in its private memory 101, and values therein for handlers of **interrupts** and non-normal **exceptions** that may occur on slave processor 25 are redirected, at step 456, to the value of an error-handler process that is a duplicate of the stack **exception** handler process for purposes of this application (An alternative to using private on-board memory is to duplicate virtual-to-physical translation tables,. . . as to provide each processor with different, exclusive, virtual-to-physical translations for certain ranges of virtual addresses.) Upon occurrence of an **interrupt** or non-normal **exception** on slave processor 25, at step 457, these values cause invocation of the handler process, at step 458. The stack **exception** and error handler processes of the slave processor 25 are communication processes that restore, at step 459, to a proper. . . been redirected at step 456, a conventional handler would have been invoked at step 458 that would have processed the **interrupt** or condition, at step 463, as is done on master processor 12.)

CLAIMS:

CLMS(6)

6. The method of claim 1 wherein the step of creating comprises the step of

corrupting **execution stack** bounds of the process executing on the first processor by operation of the one processor of the system to ensure existence of a stack bounds **exception**; and wherein the step of checking comprises the step of

checking for a stack bounds **exception** in the first processor by operation of the first processor.

CLAIMS:

CLMS(13)

13. A method of operating a multiprocessor system having a plurality of

processors, comprising the steps of:
corrupting a value of **execution stack** bounds of a user execution mode process prior to execution of the process on a slave processor of the system,. . . the user-mode process on the slave processor;
performing one of a firmware-implemented and a hardware-implemented check of a value of an **execution stack** pointer of the user process against values of the **execution stack** bounds of the user-mode process on the slave processor;
encountering a stack bounds **exception** during the check on the slave processor;
stopping execution of the user-mode process on the slave processor at the instruction, in response to the encounter,
invoking execution of a stack **exception** handler process on the slave processor, in response to the encounter;
restoring to uncorrupted values the values of the **execution stack** bounds of the user-mode process, by executing the handler process on the slave processor;
transferring the user-mode process for execution from. . .

CLAIMS:

CLMS(14)

14. . . .
the instruction of the user process on the master processor;
performing one of a firmware-implemented and a hardware-implemented check of an **execution stack** pointer of the user process against the stack bounds of the user process on the master processor; and
entering the privileged execution mode on the master processor, in response to not encountering a stack bounds **exception** during the check.

CLAIMS:

CLMS(20)

20. The system of claim 15 wherein the creating means comprise
means for corrupting **execution stack** bounds of the process executing on the first processor to ensure existence of a stack bounds **exception**; and wherein the checking means comprise
means for checking for a stack bounds **exception** in the first processor.

CLAIMS:

CLMS(26)

26. The system of claim 24 wherein the means for creating existence of an **exception** comprise
means for corrupting **execution stack** bounds of the user-mode process prior to execution of the user-mode process on the slave processor, wherein the checking and. . .

CLAIMS:

CLMS(27)

27. A multiprocessor system comprising:
a slave and a master processor;
the slave processor including
means for corrupting a value of **execution stack** bounds of a user execution mode process prior to execution of the process on the slave processor,
one of a hardware-implemented and firmware-implemented arrangement for checking a value of an **execution stack** pointer against values

of the **execution stack** bounds of the user execution mode process, in response to an attempt during execution of an instruction of the user. . . processor, and for stopping execution of the user execution mode process at the instruction and invoking execution of a stack **exception** handler process, in response to encountering a stack bounds **exception** during the check, and

means, coupled to the arrangement, for executing the handler process thereby to restore to uncorrupted values the values of the **execution stack** bounds of the user execution mode process and to transfer the user execution mode process from the slave processor to. . .

CLAIMS:

CLMS(28)

28. . . .
the transferred process at the instruction; and

one of a hardware-implemented and a firmware-implemented arrangement for checking a value of an **execution stack** pointer against values of the stack bounds of the process, in response to an attempt during execution of the instruction. . . the master processor, and for causing the privileged mode to be entered in response to not encountering a stack bound **exception** during the check.

US PAT NO: 4,597,044 [IMAGE AVAILABLE] L4: 9 of 11

DETDESC:

DETD(26)

In . . . from the processing and, execution of several different instruction codes. When an instruction program fault, instruction processing error, or an **interrupt** occurs, the collecting apparatus including units 38, 40, 42, 44, 70, 18, 47, 48 and 50 must be halted at. . . All register changes as a result of the execution in program order of instructions prior to the fault, error, or **interrupt** should be completed and any program visible register change or changes to memory as a result of execution of later. . . a valid, current copy of each of the program addressable registers to facilitate fault and error recovery and for handling **interrupts**. A record of the proper program order for all instructions in execution being processed by central processing unit 10 is. . . unloaded in proper program order; i.e., the same order or sequence in which the instructions are stored into the instruction **execution stack** 18 by the central pipeline unit's distributor 22. The instruction execution queue words contain the operation code of the instruction. . .

US PAT NO: 4,530,052 [IMAGE AVAILABLE] L4: 10 of 11

DETDESC:

DETD(27)

In . . . from the processing and, execution of several different instruction codes. When an instruction program fault, instruction processing error, or an **interrupt** occurs, the collecting apparatus including units 38, 40, 42, 44, 70, 18, 47, 48 and 50 must be halted at. . . All register changes as a result of the execution in program order of instructions prior to the fault, error, or **interrupt** should be completed and any program visible register change or changes to memory as a result of execution of later. . . a valid, current copy of each of the program addressable registers to facilitate fault and error recovery and for handling **interrupts**. A record of the proper program order for all instructions in execution being processed by central processing unit

10 is. . . unloaded in proper program order; i.e., the same order or sequence in which the instructions are stored into the instruction **execution stack** 18 by the central pipeline unit's distributor 22. The instruction execution queue words contain the operation code of the instruction. . .

US PAT NO: 4,521,851 [IMAGE AVAILABLE] L4: 11 of 11

DETDESC:

DETD(20)

In . . . from the processing and, execution of several different instruction codes. When an instruction program fault, instruction processing error, or an **interrupt** occurs, collector 36 must be halted at the end of the last successfully completed instruction. All register changes as a result of the execution in program order of instructions prior to the fault, error, or **interrupt** should be completed and any program visible register change or changes to memory as a result of execution of later. . . a valid, current copy of each of the program addressable registers to facilitate fault and error recovery and for handling **interrupts**. A record of the proper program order for all instructions in execution being processed by CPU 10 is maintained in. . . unloaded in proper program order; i.e., the same order or sequence in which the instructions are stored into the instruction **execution stack** 34 by the central pipeline unit's distributor 24. The instruction execution queue words contain the operation code of the instruction. . .

=> d his

```
     (FILE 'USPAT' ENTERED AT 16:13:11 ON 02 SEP 1999)
L1          5147 S 712/?/CCLS
L2            47 S (EXECUTION (W) STACK?)
L3          2779 S L1 AND (EXCEPTION? OR INTERRUPT?)
L4            11 S ((EXECUTION (W) STACK?) (P) (EXCEPTION? OR INTERRUPT?))

=> s l4 and java?

          1161 JAVA?
L5            1 L4 AND JAVA?

=> d 15
```

1. 5,933,635, Aug. 3, 1999, Method and apparatus for dynamically deoptimizing compiled activations; Urs Holzle, et al., 395/701 [IMAGE AVAILABLE]